A practical $O(n\log^2 n)$ time algorithm for computing the triplet distance on binary trees

Andreas Sand[1,2], Gerth Stølting Brodal[2,3], Rolf Fagerberg[4], Christian N. S. Pedersen[1,2] and Thomas Mailund*[1]

[1]Bioinformatics Research Center, Aarhus University, Denmark
[2]Department of Computer Science, Aarhus University, Denmark
[3]MADALGO, Center for Massive Data Algorithms, a Center of the Danish National Research Foundation, Denmark
[4]Department of Mathematics and Computer Science, University of Southern Denmark, Denmark

Email: Andreas Sand - asand@birc.au.dk; Gerth Stølting Brodal - gerth@cs.au.dk; Rolf Fagerberg - rolf@imada.sdu.dk; Christian N. S. Pedersen - cstorm@birc.au.dk; Thomas Mailund - mailund@birc.au.dk;

*Corresponding author

Abstract

The triplet distance is a distance measure that compares two rooted trees on the same set of leaves by enumerating all sub-sets of three leaves and counting how often the induced topologies of the tree are equal or different. We present an algorithm that computes the triplet distance between two rooted binary trees in time $O\left(n\log^2 n\right)$. The algorithm is related to an algorithm for computing the quartet distance between two unrooted binary trees in time $O\left(n\log n\right)$. While the quartet distance algorithm has a very severe overhead in the asymptotic time complexity that makes it impractical compared to $O\left(n^2\right)$ time algorithms, we show through experiments that the triplet distance algorithm can be implemented to give a competitive wall-time running time.

1 Background

Using trees to represent relationships is widespread in many scientific fields, in particular in biology where trees are used e.g. to represent species relationships, so called phylogenies, the relationship between genes in gene families or for hierarchical clustering of high-throughput experimental data. Common for these applications is that differences in the data used for constructing the trees, or differences in the computational approach for constructing the trees, can lead to slightly different trees on the same set of leaf IDs.

To compare such trees, distance measures are often used. Common distance measures include the Robinson-Foulds distance [1], the triplet distance [2], and the quartet distance [3]. Common for these three distance measures is that they all enumerate certain features of the trees they compare and count how often the features differ between the two trees. The Robinson-Foulds distance enumerates all edges in the trees and tests if the bipartition they induce is found in both trees. The triplet distance (for rooted trees) and quartet distance (for unrooted trees) enumerate all subsets of leaves of size three and four, respectively, and test if the induced topology of the leaves is the same in the two trees.

Efficient algorithms to compute these three distance measures exist. The Robinson-Foulds distance can be computed in time $O\left(n\right)$ [4] for trees with n leaves, which is optimal. The quartet distance can be computed in time $O\left(n\log n\right)$ for binary trees [5], in time $O\left(d^9\, n\log n\right)$ for trees where all nodes have degree less than

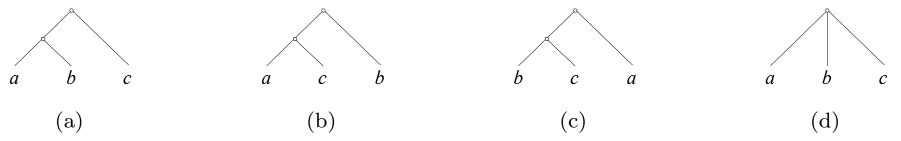


Figure 1: The four different triplet toplogies.

d [6], and in sub-cubic time for general trees [7]. See also Christiansen *et al.* [8] for a number of algorithms for general trees with different tradeoffs depending on the degree of inner nodes. For the triplet distance, $O\left(n^2\right)$ time algorithms exist for both binary and general trees [2, 9].

Brodal *et al.* [5] present two algorithms for computing the quartet distance for binary trees; one running in time $O\left(n \log n\right)$, and one running in time $O\left(n \log^2 n\right)$. The latter is the most practical, and was implemented in [10, 11], where it was shown to be slower in practice compared to a simple $O\left(n^2\right)$ time algorithm [12] unless n is above 2000. In this paper we focus on the triplet distance and develop an $O\left(n \log^2 n\right)$ time algorithm for computing this distance between two rooted binary trees. The algorithm is related to the $O\left(n \log^2 n\right)$ time algorithm for quartet distance, but its core accounting system is completely changed. As we demonstrate by experiments, the resulting algorithm is not just theoretically efficient but also efficient in practice, as it is faster than a simple $O\left(n^2\right)$ time algorithm based on [12] already for n larger than 12, and is e.g. faster by a factor of 50 when n is 2900.

2 Methods

The triplet distance measure between two rooted trees with the same set of leaf IDs is based on the topologies induced by a tree when selecting three leafs of the tree. Whenever three leaves, a, b and c, are selected, a tree can induce one of four topologies: It can either group a and b, a and c, or b and c, or it can put them at equal distance to the root, i.e. all three pairs of leaves have the same lowest common ancestor (see Fig. 1). For binary trees, the last case is not possible since this would require a node with at least three children.

The triplet distance is the number of triplets whose topology differ in the two trees. It can naïvely be computed by enumerating all $O\left(n^3\right)$ sets of three leafs and comparing the induced topologies in the two trees, counting how often the trees agree or disagree on the topology. Triplet topologies in a tree, however, are not independent, and faster algorithms can be constructed exploiting this, comparing sets of triplet topologies faster. Critchlow *et al.* [2] for example exploit information about the depth of shared ancestors of leaves in a tree to achieve an $O\left(n^2\right)$ time algorithm for binary trees while Bansal *et al.* [9] construct a table of shared leaf-sets and achieve an $O\left(n^2\right)$ time algorithm for general trees.

For the quartet distance, the analogue to the triplet distance for unrooted trees, Brodal *et al.* [5] construct an even faster algorithm by identifying sets of four different leaves in one tree through coloring all leaves with one of three different colors and then counting the number of topologies compatible with the coloring in the other tree. Using a variant of the so-called “smaller half trick” for keeping the number of different relevant colorings low, the algorithm manages to construct all relevant colorings with $O\left(n \log n\right)$ color changes. The number of topologies compatible with the coloring can then be counted in the other tree using a data structure called a “hierarchical decomposition tree”. Maintaining the hierarchical decomposition tree, however, involves a number of polynomial manipulations that, while theoretically can be done in constant time per polynomial, are quite time consuming in practice [10, 11], making the algorithm slow in practice.

A naïve algorithm that computes the quartet distance between two unrooted trees by explicitly inspecting each of the $O\left(n^4\right)$ quartets can be modified to compute the triplet distance between two rooted trees without loss of time by adding a new leaf x above the two root nodes and limit the inspection of quartets to the quartets containing this new leaf. However, the efficient algorithms for computing the quartet distance presented in [5, 7] do not explicitly inspect every quartet and therefore cannot be modified to compute the triplet distance following this simple approach.

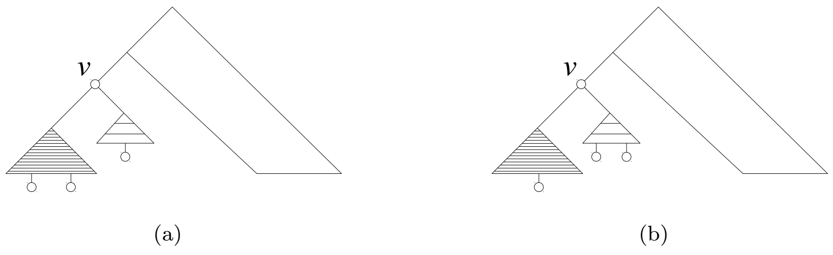


Figure 2: Coloring of a sub-tree rooted in node v lets us count all triplets rooted in v.

In the following, we develop an efficient algorithm for computing the triplet distance between two rooted binary trees T_1 and T_2 with the same set of leaf IDs. Our key contribution is to show how all triplets in one tree, say T_1, can be captured by coloring the leaves with colors, and how the smaller half trick lets us enumerate all such colorings in time $O(n \log n)$. We will then construct a hierarchical decomposition tree (HDT) for T_2 that counts its number of compatible triplets. Unlike the algorithms for computing the quartet distance [5], where the counting involves manipulations of polynomials, the HDT for triplets involves simple arithmetic computations that are efficient in practice.

2.1 Counting shared triplets through leaf colorings

A triplet is a set $\{a, b, c\}$ of three leaf IDs. For a tree T, we assign each of the $\binom{n}{3}$ triplets to the lowest common ancestor in T of the three leaves containing a, b, and c. For a node $v \in T$ we denote by τ_v the set of triplets assigned to v. Then $\{\tau_v \mid v \in T\}$ is a partition of the set $\mathcal{T}$ of triplets. Thus, $\{\tau_v \cap \tau_u \mid v \in T_1, u \in T_2\}$ is also a partition of $\mathcal{T}$. Our algorithm will find

$$\text{Shared}(\mathcal{T}) = \sum_{v \in T_1} \sum_{u \in T_2} \text{Shared}(\tau_v \cap \tau_u),$$

where Shared(S) on a set S of triplets is its number of triplets having the same topology in T_1 and T_2. The triplet distance of T_1 and T_2 is then $\binom{n}{3} - \text{Shared}(\mathcal{T})$.

In the algorithm, we capture the triplets τ_v by a coloring of the leaves: if all leaves not in the subtree of v have no color, all leaves in one sub-tree of v are "red", and all leaves in the other sub-tree are "blue", then τ_v is exactly the triplets having two leaves colored "red" and one leaf colored "blue", or two leaves colored "blue" and one leaf colored "red". See Fig. 2.

For such a coloring according to a node $v \in T_1$, and for a node $u \in T_2$, the number Shared$(\tau_v \cap \tau_u)$ could be found as follows: let x and y be the two subtrees of u, let $x(r)$ and $y(r)$ be the number of leaves colored red in the subtrees x and y, respectively, and $x(b)$ and $y(b)$ the number of leaves colored blue. The number Shared$(\tau_v \cap \tau_u)$ is then $\binom{x(b)}{2} \cdot y(r) + \binom{y(b)}{2} \cdot x(r) + \binom{y(r)}{2} \cdot x(b) + \binom{x(r)}{2} \cdot y(b)$. We call these the triples of u compatible with the coloring.

Explicitly going through T_1 and coloring for each node v would take time $O(n)$ per node, for a total time of $O(n^2)$. We reduce this to $O(n \log n)$ by using the smaller half trick. Going through T_2 for each coloring and counting the number of compatible triplets would also take time $O(n^2)$. Using a HDT we find this count in $O(1)$ time, while updating the structure takes time $O(\log n)$ after each leaf color change. The result is $O(n \log^2 n)$ total running time. In essence, the HDT performs the inner sum of Shared$(\mathcal{T}) = \sum_{v \in T_1} \sum_{u \in T_2} \text{Shared}(\tau_v \cap \tau_u)$, while the coloring algorithm performs the outer sum.

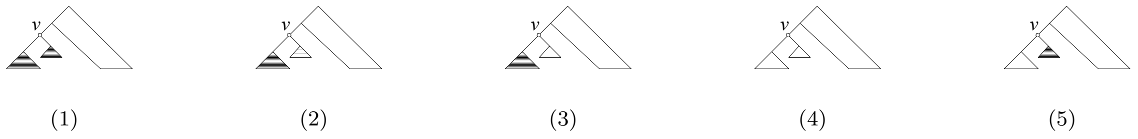


Figure 3: The five steps of the coloring in the smaller-half trick.

2.2 Smaller half trick

We go through nodes v in a depth first order while maintaining two invariants of the algorithm: 1) Before we process v, the entire subtree of v is colored "red" and the rest of the tree has no color; 2) When we return from the depth first recursion, the entire tree has no color.

For v let $S(v)$ denote the smallest subtree of v and let $L(v)$ denote the largest subtree of v. We go through the coloring as follows (see Fig. 3):

1. Color $S(v)$ "blue". Now v has the coloring that enable us to count the triplets for v.
2. Remove the color for $S(v)$. Now we can call recursively on $L(v)$ while satisfying the invariant.
3. Returning from the recursive call, the entire tree is colorless by invariant 2.
4. Color $S(v)$ "red". Now we satisfy invariant 1 for calling recursively on $S(v)$.
5. Call recursively on $S(v)$. When we return we satisfy invariant 2 for returning from the recursive call.

Using this recursive algorithm, we go through all colorings of the tree. In each instance (not counting recursive calls), we only color leaves in $S(v)$, and only a constant number of times. Thus, a leaf ℓ is only colored when visiting an ancestor node v where $\ell \in S(v)$, i.e. ℓ is in the smaller subtree of v. Since ℓ can have at most $O(\log n)$ such ancestors, each leaf will only be colored at most $O(\log n)$ times, implying a total of $O(n \log n)$ color changes.

2.3 Hierarchical decomposition tree

We build a data structure, the *hierarchical decomposition tree* (HDT), on top of the second tree T_2 in order to count the triplets in T_2 compatible with the coloring of leaves in the first tree T_1. The HDT is a balanced binary tree where each node corresponds to a connected part of T_2. Each node in the HDT, or *component*, keeps a count of the number of compatible triplets the correspondent part of T_2 contains, plus some additional book-keeping that makes it possible to compute this count in each component in constant time using the information stored in the component's children in the HDT.

The HDT contains three different kinds of components:

- **L**: A leaf in T_2,
- **I**: An inner node in T_2,
- **C**: A connected sub-part of T_2,

where for type **C** we require that at most two edges in T_2 crosses the boundary of the component; at most one going up towards the root and at most one going down to a subtree.

The leaves and inner nodes of T_2 are transformed into **L** and **I** components, respectively, and constitute the leaves of the HDT. **C** components are then formed by pairwise joining other components along an edge in T_2 by one of two compositions, see Fig. 5. **C** components can be thought of as consisting of a path from a sub-tree below the **C** component going up towards the root of T_2, such that all trees branching off to other children along the path are all contained in the component. In the following we show how the HDT of T_2 can be constructed in time $O(n)$, and we prove that the height of the HDT is $O(\log n)$.

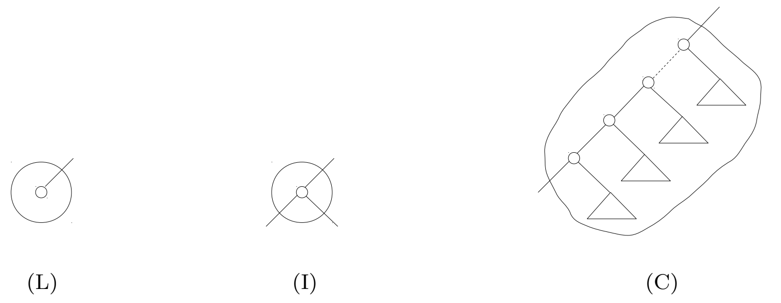


Figure 4: The three different types of components. **L** and **I** components contain a single node from the underlying tree while **C** components contain a connected set of nodes.

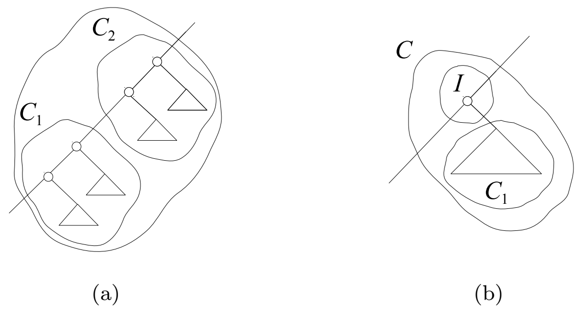


Figure 5: The two different ways of constructing a **C** component by merging two underlying components. The topmost of the components can either be a **C** component (a) or an **I** component (b) while the bottommost component, C_1, must be a **C** or **L** component. If the topmost component is an **I** component, the bottommost must be downwards closed, i.e. it cannot have a downwards edge crossing its boundary.

The construction algorithm operates on *components* and *edges*. Each component is of one of the types **L**, **I**, or **C**. It has a *parent* pointer, pointing to its parent component in the HDT, and a bit, *down_closed*, indicating if the component corresponds to a complete subtree in the original tree. When a component does not yet have a parent in the HDT, we make the *parent* pointer point to the component itself. We use this pointer to test if an edge has been contracted in the HDT construction. Edges consist of two pointers, an *up* pointer and a *down* pointer that points to the components above and below the edge in T_2.

In a single traversal of the tree, the algorithm initially builds a component for each node in the tree (an **L** component for each leaf and an **I** component for each inner node) and an edge for each edge in the tree. The *parent* pointer of each component is initially set to point to the component itself, and the *down_closed* bit is set to true for **L** components and false for **I** components. The edges are put in a list *es*. We then perform a series of iterations, each constructing one level of the HDT. In each iteration, edges whose neighbors can be joined to form a **C** component via the constructions in Fig. 5 are greedily removed from *es*, and another list, *next*, is used to keep the edges that cannot be contracted in this iteration. The details of an iteration is shown in Fig. 6. The process stops when *es* becomes empty.

In case 1, one of *e's* neighbors already has a parent, thus this neighbor has already been contracted into a **C** component in this iteration and should not be contracted again. Case 2 is the situation in Fig. 5(a), and case 3 is the situation in Fig. 5(b). In case 4, *e.down* is an **I** component or *e.up* is an **I** component and *e.down* does not correspond to a complete subtree, hence none of the constructions in Fig. 5 apply. After removing all edges from *es*, the *up* and *down* pointers of the remaining edges in *next* are updated in

```
 1 for each edge e in es
 2     if e.up.parent ≠ e.up or e.down.parent ≠ e.down
 3         /* Case 1: At least one of the end-points of e is already contracted */
 4         move e from es to next
 5     else if e.up.type = C and (e.down.type = C or e.down.type = L)
 6         /* Case 2: Fig. 5(a). */
 7         remove e from es
 8         make a new component c
 9         c.down_closed = e.down_closed
10         e.up.parent = e.down.parent = c
11     else if e.up.type = I and e.down.down_closed = True
12         /* Case 3: Fig. 5(b). */
13         remove e from es
14         make a new component c
15         c.down_closed = False
16         e.up.parent = e.down.parent = c
17     else
18         /* Case 4 */
19         move e from es to next
20 for each edge e in next
21     e.up = e.up.parent
22     e.down = e.down.parent
23     move e from next to es
```

Figure 6: Algorithm for constructing the hierarchical decomposition tree. The listing shows the algorithm run for each level of the HDT construction. This algorithm is repeated until the *es* list is empty.

lines 21–22, such that they point to either a newly created component or still point to the same component. The edges in *next* are finally moved back to *es* in line 23, and a new iteration is started. The algorithm finishes when *es* is empty, and the root of the HDT is the **C** component resulting from joining the ends of the last edge.

We now argue that height of the HDT is $O(\log n)$, and that the construction time is $O(n)$. Each iteration of the code in Fig. 6 takes time linear in the number $E = |es|$ of edges remaining to be contracted at the beginning of the iteration. The height and time bounds follow if we can argue that the number of edges decreases geometrically for each iteration. In the following we argue that the number of edges after one iteration is at most $11/12 \cdot E$.

We first argue that the number of contractible edges at the beginning of the iteration is at least $E/4$. Note that only edges incident to **I** components might not be contractible, and that the number of down-closed components is at least one larger than the number of **I** components. If the number of **I** components is at most $E/4$, then at most $3 \cdot E/4$ incident edges might not be contractible, i.e. at least $E/4$ edges are contractible. Otherwise the number of **I** components is more than $E/4$, and therefore the number of down-closed components is more than $E/4+1$. Since the parent edges from all down-closed components are contractible (for $E \geq 1$), the number of contactable edges is again at least $E/4$.

Since each contracted edge can prevent at most two other edges incident to the two merged components (see Fig. 5) from being contracted in the same iteration, each iteration will contract at least $1/3$ of the at least $E/4$ contractible edges. It follows that an iteration reduces the number of contractible edges by at least $E/12$.

2.4 Counting triplets in the hierarchical decomposition tree

In each component we keep track of N, the number of triplets contained within the component (i.e., where the three leaves of the triplet are within the component) that are compatible with the coloring. When we change the coloring, we update the HDT to reflect this, so we can always read off the total number of compatible triplets from the root of the HDT in constant time.

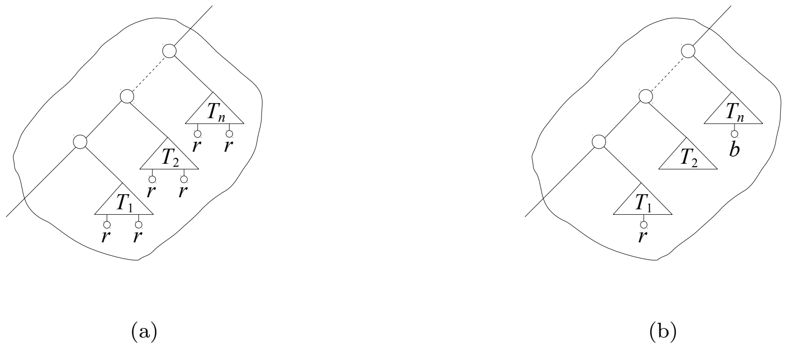


Figure 7: The two cases of counting triplets in a **C** component.

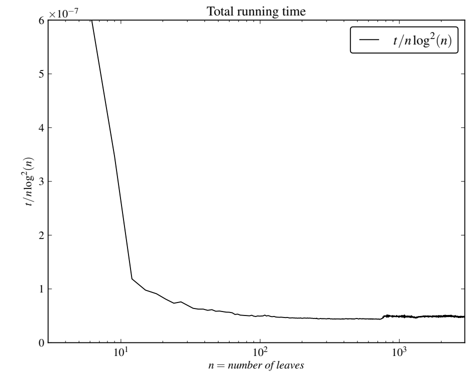
By adding a little book-keeping information to each component we make it possible to compute N (and the book-keeping information itself) for a component in constant time from the information in the component's children in the HDT. This has two consequences: it makes it possible to add the book-keeping and N to all components after the construction in linear time, and it makes it possible to update the information when a leaf changes color by updating only components on the path from the leaf-component to the root in the HDT, a path that is bounded in length by $O(\log n)$. Since we only change the color of a leaf $O(n \log n)$ times in the triplet distance algorithm, it is this property of the HDT that keeps the total running time at $O(n \log^2 n)$. For the book-keeping we store six numbers in each component in addition to N:

- R: The number of leaves colored red in the component.
- B: The number of leaves colored blue in the component.
- $\widetilde{rr}$: The number of pairs of red leaves found in the same sub-tree of a **C** component. Let T_i, $i = 1, \ldots, n$, denote the sub-trees in the component, see Fig. 7(a), and let $r(i)$ denote the number of red leaves in tree T_i. Then $\widetilde{rr} = \sum_{i=1}^{n} \binom{r(i)}{2}$.
- $\widetilde{bb}$: The number of pairs of blue leaves found in the same sub-tree of a **C** component.
- $\widehat{rb}$: The number of pairs of leaves with a red leaf in one sub-tree and a blue in another sub-tree, where the red leaf is in a tree further down on the path in a **C** component. Let T_i, $i = 1, \ldots, n$, denote the sub-trees in the component, see Fig. 7(b), and let $r(i)$ denote the number of red leaves in tree T_i and $b(i)$ the number of blue leaves in T_i. Then $\widehat{rb} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} r(i) \cdot b(j)$.
- $\widehat{br}$: The number of pairs of leaves with a red leaf in one sub-tree and a blue in another sub-tree, where the blue leaf is in a tree further down on the path in a **C** component.

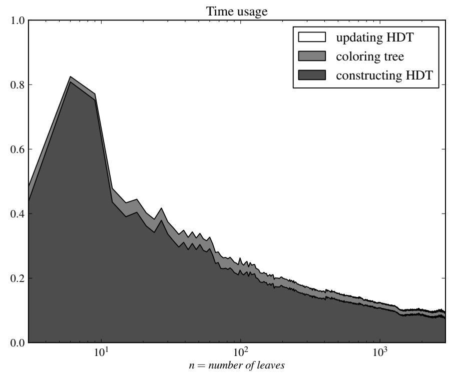
We describe how the book-keeping variables and N are computed through a case-analysis on how the components are constructed. **L** and **I** components are constructed in only one way, while **C** components are constructed in one of two ways (see Fig. 5).

L components: For a leaf component, R is 1 if the leaf is colored red and 0 otherwise, B is 1 if the leaf is colored blue and 0 otherwise, and all other counts are 0.

I components: All counts are 0.



(a) Total running time divided by $n \log^2 n$.



(b) The percent of time used on the three parts of the algorithm.

Figure 8: The left figure shows the total running time divided by $n \log^2 n$ showing that the theoretical running time is achieved in the implementation. The right figure shows the percent of time used on the three parts of the algorithm: constructing the HDT of the second tree, coloring the leaves in the first tree, and updating the HDT accordingly. Constructing the HDT takes a considerable part of the time, but as the trees grow, updating the HDT takes a larger part. The plots show the average over 50 experiments for each size n.

C components, case Fig. 5(a): Let x be one of the counters R, B, $\widetilde{rr}$, ... listed above for a **C** component, and let $x(1)$ and $x(2)$ denote the corresponding counter in component C_1 and C_2, respectively, with C_2 above C_1 in the underlying tree. Then

$$\begin{aligned} R &= R(1) + R(2) & B &= B(1) + B(2) \\ \widetilde{rr} &= \widetilde{rr}(1) + \widetilde{rr}(2) & \widetilde{bb} &= \widetilde{bb}(1) + \widetilde{bb}(2) \\ \widehat{rb} &= \widehat{rb}(2) + \widehat{rb}(1) + R(1) \cdot B(2) & \widehat{br} &= \widehat{br}(2) + \widehat{br}(1) + B(1) \cdot R(2) \ . \end{aligned}$$

The triplet count is then computed as

$$\begin{aligned} N &= N(1) + N(2) + \binom{R(1)}{2} \cdot B(2) + \binom{B(1)}{2} \cdot R(2) \\ &+ \widetilde{rr}(2) \cdot B(1) + \widetilde{bb}(2) \cdot R(1) + R(1) \cdot \widehat{rb}(2) + B(1) \cdot \widehat{br}(2) \ . \end{aligned}$$

C components, case Fig. 5(b): Let $x(1)$ denote one of the counters listed above for component C_1. Then $R = R(1)$, $B = B(1)$, $\widetilde{rr} = \binom{R(1)}{2}$, $\widetilde{bb} = \binom{B(1)}{2}$, $\widehat{rb} = \widehat{br} = 0$. Since the inner node in the composition does not contain any leaves, the triplet count is simply $N = N(1)$.

3 Results and discussion

We implemented the algorithm in C++ and a simple $O(n^2)$ time algorithm to ensure that it computes the correct triplet distance.

We then verified the running time of our algorithm, see Fig. 8(a). As seen in the figure, the running time evens out when we divide with $n \log^2 n$ giving us confidence that the analyzed running time is correct.

We also measured where in the algorithm time was spent, whether it is in constructing the HDT, in coloring the leaves in the first tree, or in updating the counts in the HDT. Figure 8(b) illustrates the time spend on each of these three parts of the algorithm, normalized so the running time sums to one. For

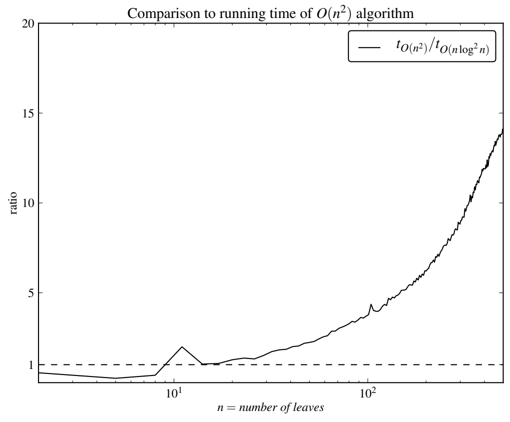


Figure 9: Ratio between the running times for the $O(n^2)$ time algorithm and the $O(n\log^2 n)$ time algorithm.

small trees, constructing the HDT makes up a sizable fraction of the running time, not surprising since the overhead in constructing components is larger than updating them. As the size of the trees increase, more time is spent on updating the HDT, as expected since updating the HDT runs in $O\left(n\log^2 n\right)$ while the other operations are asymptotically $O\left(n\log n\right)$.

When changing the color of leaves, we spent time $O\left(\log n\right)$ updating the book-keeping in the HDT for each leaf. We only count after a complete sub-tree has changed color, however, so instead of updating the HDT for each color-change we could just mark which leaves have changed color and then update the HDT bottom-up, so each inner node would only be updated once when it is on a path from a changed leaf. We implemented this, but found that the extra book-keeping from marking nodes and then updating increased the running time by 10%–15% compared to just updating the HDT.

To render the use of the algorithm in practice, we implemented an efficient $O(n^2)$ time algorithm based on the quartet distance algorithm presented in [12]. Figure 9 shows the ratio of the running time for the $O(n^2)$ time algorithm against the $O(n\log^2 n)$ time algorithm. It is evident that our algorithm is fastest for all practical purposes. The speed-up factor for $n = 2900$ is 46.

4 Conclusions

We have presented an $O\left(n\log^2 n\right)$ time algorithm for computing the triplet distance between two binary trees and experimentally validated its correctness and time analysis.

The algorithm builds upon the ideas in the $O\left(n\log^2 n\right)$ time algorithm for computing the quartet distance between binary trees [13], but where the book-keeping in the quartet distance algorithm is rather involved, making it inefficient in practice, the book-keeping in the triplet distance algorithm in this paper is entirely different, and significantly simpler and faster.

Compressing the HDT during the algorithm makes it possible to reduce the running time of the quartet distance algorithm to $O\left(n\log n\right)$ and the same approach can also reduce the running time of the triplet algorithm to $O\left(n\log n\right)$. We have left for future work to experimentally test whether this method incurs too much overhead to make it practically worthwhile.

Unlike the $O\left(n^2\right)$ time algorithm of Bansal *et al.* [9] our algorithm does not generalize to non-binary trees. It is possible to extend the algorithm to non-binary trees by employing more colors, as was done for the quartet distance [6], but this makes the algorithm depend on the degree of nodes, and future work is needed to develop a sub-quadratic algorithm for general rooted trees.

Author's contributions

All authors contributed to the design of the presented algorithm. AS implemented the data structures and the algorithm. AS, CNSP, and TM designed the experiments, and AS conducted these. All authors have contributed to, seen and approved the manuscript.

References

1. Robinson DF, Foulds LR: **Comparison of Phylogenetic Trees**. *Mathematical Biosciences* 1981, **53**:131–147.
2. Critchlow DE, Pearl DK, Qian CL: **The triples distance for rooted bifurcating phylogenetic trees**. *Systematic Biology* 1996, **45**(3):323–334.
3. Estabrook GF, McMorris FR, Meacham CA: **Comparison of Undirected Phylogenetic Trees Based on Subtrees of Four Evolutionary Units**. *Systematic Zoology* 1985, **34**(2):193.
4. Day WHE: **Optimal-Algorithms for Comparing Trees with Labeled Leaves**. *Journal of Classification* 1985, **2**:7–28.
5. Brodal GS, Fagerberg R, Pedersen CNS: **Computing the quartet distance between evolutionary trees in time $O(n \log n)$**. *Algorithmica* 2004, **38**(2):377–395.
6. Stissing MS, Pedersen CNS, Mailund T, Brodal GS, Fagerberg R: **Computing the quartet distance between evolutionary trees of bounded degree**. In *Proceedings of the 5th Asia-Pacific Bioinformatics Conference (APBC)*, Imperial College Press 2007:101–110.
7. Nielsen J, Kristensen A, Mailund T, Pedersen CNS: **A sub-cubic time algorithm for computing the quartet distance between two general trees**. *Algorithms for Molecular Biology* 2011, **6**:15.
8. Christiansen C, Mailund T, Pedersen CNS, Randers M, Stissing MS: **Fast calculation of the quartet distance between trees of arbitrary degree**. *Algorithms for Molecular Biology* 2006, **13**.
9. Bansal MS, Dong J, Fernández-Baca D: **Comparing and aggregating partially resolved trees**. *Theoretical Computer Science* 2011, **412**(48):6634–6652.
10. Mailund T, Pedersen CNS: **QDist–quartet distance between evolutionary trees**. *Bioinformatics* 2004, **20**(10):1636–1637.
11. Stissing MS, Mailund T, Pedersen CNS, Brodal GS, Fagerberg R: **Computing the all-pairs quartet distance on a set of evolutionary trees**. *Journal of Bioinformatics and Computational Biology* 2008, **6**:37–50.
12. Bryant D, Tsang J, Kearney P, Li M: **Computing the quartet distance between evolutionary trees**. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics 2000:285–286.
13. Brodal GS, Fagerberg R, Pedersen CNS: **Computing the quartet distance between evolutionary trees in time $O(n \log^2 n)$**. In *Proceedings of the 12th International Symposium on Algorithms and Computation (ISAAC), Volume 2223 of* Lecture Notes in Computer Science, Springer 2001:731–742.